The University of North Carolina at Chapel Hill

Senior Honors Thesis

Department of Computer Science

Exploring the Efficiency of First-Order Proving Methods

Author:
Mark Molinaro

Supervisor:
Dr. David Plaisted

April 26, 2017

Abstract

Many automated theorem proving applications rely on the DPLL algorithm for deciding the satisfiability of a set of propositional logic formulae. For first-order logic formulae, ground clauses within the Herbrand universe may be exhaustively enumerated below an incrementing size-bound and fed as input to DPLL. From even a cursory investigation of these enumerated clauses, it is evident that many of them have multiple repeated terms. Here, we explore a potential method for exploiting the size-bound by "cheating in" larger clauses with many repeating terms that may be relevant to the proof.

1 Introduction

Automated theorem proving is all about efficiency. The first proposed provers could never realistically be used in any meaningful capacity because they would have been too inefficient. Over the past few decades, new and improved techniques have enabled such provers to be effectively implemented for real applications. Even so, further improvements in efficiency would boost both the performance and viability of those current applications. Here, we will start with a survey of automated theorem proving history, applications, and methods. Then I will present a previously unexamined method for potentially improving the efficiency of first-order logic provers. Though this method may not be entirely feasible, understanding the underlying idea and implementation problems may provide insight for future exploration.

2 Background

Before diving into the particulars of the project, it is important to understand both the history and applications of automated theorem proving to get a more wholistic understanding of the importance of the subject and its development.

2.1 History

Formal logic had its founding in Greco-Roman times with philosophers such as Aristotle. Focused primarily on the concept of deduction, Aristotle authored multiple works outlining his explorations with formal logics. A col-

lection of his works, entitled *The Oraganon* [13], details much of the same concepts still taught today, though with differing notation.

Moving quite a ways forward in history to the late 19^{th} century, Gottlob Frege [5] and Giuseppe Peano [14] introduced early modern logical notations. In 1921, Emil Post [16] further formalized the concepts and notations of propositional logic. Later, in 1960, John McCarthy [9] presented a hypothetical "Advice Taker" to give and receive advice represented declarative (first-order) knowledge using predicate notation and relations.

Around the same time, many individuals started to conceptualize and develop automated proving techniques. Gilmore [7], Prawitz [17], and Davis and Putnam [4] developed proving methods by exhaustively checking all ground clauses based on Herbrand's theorem (a topic we will revisit later). These initial attempts shown inefficient, Robinson [18] developed resolution-based proving methods which utilized the power of unification and the resolution inference rule. The Argonne group first implemented resolution-based provers with much success. While resolution methods seemed to thrive at first, many inefficiencies were discovered that researchers were unable to fully overcome.

Various approaches were attempted in order to improve the efficiency of proving procedures, giving rise to powerful languages and provers such as Prolog, Vampire [8], Otter [10], its successor Prover9 [12], and various higher-order and interactive provers.

2.2 Applications

Interactive provers supplement a human's endeavor to construct a proof. In such systems, an individual will supplement the proof search by (heuristically) determining intermediate steps to guide procedure in the right direction. Non-interactive provers are simply given a starting set of axioms and a goal statement to prove by manipulations of those axioms. Both classes of provers have found to be very useful in a variety of applications, including mathematics, formal verification, and software development.

2.2.1 Robbins Problem

Theorem provers have been found to be an effective tool for finding proofs in various areas within mathematics. One such example is Robbins equation.

The basis for Boolean algebra consists of commutativity, associativity, and the Huntington equation.

$$x \vee y \iff y \vee x \text{ [commutativity]}$$
$$(x \vee y) \vee z \iff x \vee (y \vee z) \text{ [associativity]}$$
$$\neg(\neg x \vee y) \vee \neg(\neg x \vee \neg y) \iff x \text{ [Huntington equation]}$$

Herbert Robbins conjectured that the Huntington equation could be replaced by the following, known as Robbins equation.

$$\neg(\neg(x \vee y) \vee \neg(x \vee \neg y)) \iff x \text{ [Robbins equation]}$$

The problem was to determine whether every Robbins algebra was Boolean. For many years, it stood only as conjecture. Neither Huntington, Robbins, nor anyone else could find a proof until 1997 when William McCune [11] finally presented one. McCune was able to find the solution with a successful search by the EQP theorem prover, which completed its search in about 8 days.

2.2.2 Verification

ATP methods have shown to be vital in formal hardware and software verification. As very-large-scale integration (VLSI) design became prominent, a formal method for ensuring correctness in chips became essential. These automated methods allowed developers to test their designs despite the impossibility of manually testing for every possible set of inputs. Had such methods been more developed and widely used earlier, the Pentium FDIV floating-point bug [1] may have been caught and prevented.

2.2.3 Software Development

An application of theorem proving in software development is the Amphion system [20], developed for use at NASA. This system allowed end-users to develop graphical specifications that are converted to first-order formulas and proved by reusable subroutines under the hood. This process consists of three steps: building a graphical representation, program synthesis, then a production of the final program into the target language — Fortran77, in this case. Such systems aided researchers and engineers in their work without such strict prerequisites for knowledge of computer programming.

3 Definitions

3.1 Propositional Logic

Propositional logic is the branch of logic that describes simple propositions and the compositions of such. First, we will define the syntax used when discussing and reasoning with propositional logic.

Propositional Signature – A (nonempty) set of symbols, called **atoms** (or variables). This set should contain all objects on which we can do our reasoning.

Connectives – Formulas are formed from the composition of atoms and connectives. The connectives available are as follows:

- 0-place Connectives: $\top, \bot$
- Unary Connective: $\neg$
- Binary Connectives $\vee, \wedge, \rightarrow, \leftrightarrow$

Well-Formed Formula (wff) – A Well Formed Formula can be recursively defined as follows:

1. Any atom, $\top$, or $\bot$ is a well-formed formula.
2. If $\aleph$ is a well-formed formula, $\neg\aleph$ is a well-formed formula.
3. If $\aleph$ and β are both well-formed formula, $\aleph \oplus \beta$ is a well-formed formula, where $\oplus$ is one of the binary connectives listed above.
4. Nothing that cannot be constructed by steps 1-3 is a well-formed formula.

Interpretations – An interpretation of a formula F is a mapping of F to a truth value, $true$ or $false$. The 0-place connectives $\top$ and $\bot$ always map to $true$ and $false$, respectively. Individual atoms can be interpreted as either $true$ or $false$. To map an arbitrarily complex formula to a truth value, a similarly recursive definition to that of *wff*s can be used, following the base truth values of each connective of *wff*s below.

Given F, every possible interpretation can be mapped out in a truth table. If there is some interpretation such that $F = true$, that interpretation satisfies F and F is satisfiable. If there is no such interpretation, F is unsatisfiable. If every interpretation of $F = true$, then F is a tautology.

Table 1: Connective Interpretations

x	y	$\neg x$	$x \wedge y$	$x \vee y$	$x \rightarrow y$	$x \leftrightarrow y$
false	false	true	false	false	true	true
false	true	-	false	true	true	false
true	false	false	false	true	false	false
true	true	-	true	true	true	true

A set of formulas (or knowledge base) Γ is satisfiable if there exists at least one interpretation that satisfies all formulas within Γ. If every interpretation that satisfies Γ also satisfies another formula F, then Γ entails F ($\Gamma \models F$).

Γ can be extended by an explicit definition to include new formulas – using atoms within the original signature that were otherwise not part of the knowledge base. The extended knowledge base is satisfiable iff the original knowledge base is satisfiable.

Natural Deduction – Natural Deduction is a process of deriving formulas (or sequents) from a finite knowledge base Γ. A formula F can be derived from Γ using the inference rules of propositional logic, described, in German, by Gentzen [6]. These can loosely be described as a group of rules for introduction, a group of rules for elimination, a contradiction rule, and a weakening rule. Sequents can be discovered by iteratively applying these rules to formulas within a knowledge base.

This deductive system is sound and complete. A logical system is sound iff it only yields formulas that are valid under the system. A logical system is complete if every formula that is valid can be derived by the set forth inferences.

Formulas can be proved under Γ by constructing or chaining together axioms, formulas in Γ, and the sequents implied by it. Once a sequent is constructed, it may be used further in the proof as if it is part of the original knowledge base.

3.2 First-Order Logic

First-Order Logic is an extension on the propositional signature that includes function symbols, constant symbols, quantifiers, and predicate constants. For brevity, the syntax and semantics of these symbols are presented together.

Function Symbols – Functions of arity n take n elements as parameters

and returns an element in the domain. As an example $mother\_of(x)$, a 1-ary function, takes an element in the domain of discourse as input and returns another.

Constant Symbols – An individual constant symbol represents a fixed element in the domain. These can also be interpreted as 0-ary functions.

Logical Quantifiers – Quantifiers are used to specify the number of elements within the domain of discourse that satisfy a formula. The two traditional quantifiers are the *universal* ("for all") $\forall$ quantifier and the *existential* ("there exists") $\exists$ quantifier. Quantification is restricted only to objects; the quantification of formulae and sets of objects exists only in higher-order logics.

Predicate Constants – Predicates, also with an arity n, take n elements as parameters and represents a relationship between those elements. Instead of an element, predicates output a truth value. As an example, $are\_friends(Mark, Matt)$ will either return $true$ or $false$, depending on the friendship status of Mark and Matt.

Terms – Terms can be recursively defined as follows:

1. Any variable is a term.
2. Any constant is a term.
3. If $t_1, ..., t_n$ are terms, and f is a function symbol of arity n, then $f(t_1, ..., t_n)$ is a term.
4. Nothing that cannot be constructed by steps 1-3 is a well-formed formula.

Formulae – Formulae can be recursively defined as follows:

1. If $t_1, ..., t_n$ are terms, and P is a predicate constant of arity n, then $P(t_1, ..., t_n)$ is a formula.
2. If $\aleph$ is a formula, then $\neg\aleph$ is a formula.
3. If $\aleph$ and β are both formulae, then $\aleph \oplus \beta$ is a formula, where $\oplus$ is one of the binary connectives.
4. If $\aleph$ is a formula and x is a variable, then $\exists x\aleph$ and $\forall x\aleph$ are formulae.
5. Nothing that cannot be constructed by steps 1-4 is a formula.

Formulae of the form dictated in Rule 1 are called **atomic formulae**. A **literal** is an atomic formula or its negation. If a variable x is within the scope of a quantifier ($\forall x$ or $\exists x$), x is **bound**, otherwise, it is **free**. A formula is a **sentence** if it contains no free variables. For a sentence F and interpretation I, F^I is a truth value assigned by I to F.

Just as with propositional logic, deduction can be done with first-order formulas using a set of inference rules (modus ponens & universal generalization) and further identities. A sentence is provable in first-order logic if it can be derived from from successive applications of axioms. A sentence F is provable with respect to a set of statements $\{a_1, ..., a_n\}$ if $\bigwedge a_i \to F$ is provable in first-order logic.

4 Clausal Form

In order to utilize mechanical proving techniques, most algorithms require first-order formulas to first be converted to clausal form in order to more uniformly store and manipulate statements. Converting an arbitrary first-order formula requires five steps.

For this section, we will follow a formula of the form:

$$\neg\forall x\neg(F(x) \wedge \neg F(f(x)))$$

to the standardized clausal form.

4.1 Pushing Negations

The first step in the conversion is to push negations inward. To do this, we follow a set of negation pushing rules. The rules are applied iteratively until no more replacements can be made.

$$(A \leftrightarrow B) \to (A \to B) \wedge (B \to A)$$
$$(A \to B) \to ((\neg A) \vee B)$$
$$\neg\neg A \to A$$
$$\neg(A \wedge B) \to (\neg A) \vee (\neg B)$$
$$\neg(A \vee B) \to (\neg A) \wedge (\neg B)$$
$$\neg\forall x A \to \exists x(\neg A)$$

$$\neg\exists x A \to \forall x(\neg A)$$

These rules are fairly obvious, as the first two are the definitions of $\leftrightarrow$ and $\to$, and the last four are De Morgan's laws and first-order extensions of such.

In our example, we can first apply generalized De Morgan's $\neg\forall$ rule, then the double $\neg$ rule:

$$\neg\forall x\neg(F(x) \land \neg F(f(x)))$$
$$\exists x\neg\neg(F(x) \land \neg F(f(x)))$$
$$\exists x(F(x) \land \neg F(f(x)))$$

Moving forward to the next section, our formula stands as:

$$\exists x(F(x) \land \neg F(f(x)))$$

4.2 Pulling Universal Quantifiers

The next step is to pull all universal quantifiers to the front. Similar to pushing negations, we can achieve this by iteratively applying the following rules until no more can be applied:

$$(\forall x A) \lor B \to \forall x(A \lor B)$$
$$A \lor (\forall x B) \to \forall x(A \lor B)$$
$$(\forall x A) \land B \to \forall x(A \land B)$$
$$A \land (\forall x B) \to \forall x(A \land B)$$

Our example contains no universal quantifiers. As none of the above rules apply, this step is done with no modifications to the formula.

$$\exists x(F(x) \land \neg F(f(x)))$$

4.3 Skolem Functions

The third step in the conversion process requires replacing each existentially quantified variables by Skolem functions. To be more precise:

$$\exists x P(x) \rightarrow P(f(x_1, ..., x_n))$$

where f is a newly contrived (and uniquely named) Skolem function with parameters including all universally quantified variables within P's scope. If there are no variables outside of x, the Skolem function is of arity 0, or simply a Skolem constant. As a note, this conversion preserves the satisfiability of the formula – that is, the new formula is satisfiable iff the old formula is satisfiable – but may not preserve equivalence. As our proving algorithm is based on determining satisfiability (discussed in more detail later), equivalency is not required.

Using our example, the only variable x is existentially quantified. We replace the existential quantifier with a new Skolem constant a.

$$F(a) \wedge \neg F(f(a))$$

4.4 Conjunctive Normal Form

Conjunctive normal form (CNF) is a formula structured as a conjunction of disjunctions of literals, such as:

$$(A \vee B \vee \neg C) \wedge (\neg A \vee B \vee C) \wedge (\neg A \vee \neg B \vee \neg C)$$

Normal forms such as this are useful as a standardized way of uniquely representing a formula. Converting propositional formulas with n clauses into CNF can, in the worst case, produce 2^n clauses. There are methods to potentially reduce the size of these formulas, including the addition of extra predicate symbols for sub-formulas – but such methods are not discussed here.

Conversion to CNF is done by iteratively distributing "or's over and's" until the formula satisfies the conditions for CNF. The rules, specifically, are:

$$(A \vee (B \wedge C)) \rightarrow (A \vee B) \wedge (A \vee C)$$
$$((A \wedge B) \vee C)) \rightarrow (A \vee B) \wedge (A \vee C)$$

Fortunately, our example is already in CNF, with each disjunction containing only one literal. We can rearrange parentheses to emphasize the structure.

$$(F(a)) \wedge (\neg F(f(a)))$$

4.5 Clausal Form

Clausal form is simply a syntactic change from CNF. Once in CNF, all universal quantifiers are removed – as all variables at that point are either implicitly or explicitly universally quantified – and the Boolean connectives are replaced in favor a structuring as a set of sets of literals. As an example,

$$(A_1 \vee A_2) \wedge (B_1 \vee B_2) \wedge (C_1 \vee C_2)$$

becomes

$$\{\{A_1, A_2\}, \{B_1, B_2\}, \{C_1, C_2\}\}$$

In our example, we finish our conversion to clausal form with:

$$\{\{F(a)\}, \{\neg F(f(a))\}\}$$

5 Mechanical Proving

Now that our input first-order formula is in the proper form, we will discuss proving techniques, specifically focusing on instance-based method utilizing DPLL.

5.1 Herbrand Interpretations

A **Herbrand Interpretation** I is a special type of interpretation defined relative to a set of clauses. The domain of I consists of all finite terms constructible by the available constant symbols (adding an arbitrary constant if none exists) and functions. Intuitively, a Herbrand Interpretation consists of an enumeration of every valid permutation of constants and functions, forming only ground terms. It should be noted that these interpretations do not include predicates.

Using our prior example, we can construct a set of constant symbols S and functions symbols F with subscripted arity. As there is now a constant a in our formula, we no not need to add an arbitrary constant.

$$S : \{a\}$$
$$F : \{f_1, g_1\}$$

From this we can construct the domain as such:

$$D : \{a, f(a), g(a), f(f(a)), g(f(a)), f(g(a)), g(g(a)), ...\}$$

These interpretations are specifically interesting for mechanical theorem proving because to determine the satisfiability of a set of clauses, we only need to consider ground clauses constructed from the Herbrand Interpretations. This is because a set of clauses C is satisfiable iff there is a Herbrand Interpretation I such that $I \models C$.

A **substitution** is defined as a mapping from variables to terms. For example, we can use the substitution $\{x \vdash a\}$ on the formula $P(f(x), x)$ to yield $P(f(a), a)$. All instances of the variable x are replaced with a. In this case, we substituted in a ground term, but it is not necessary to do so – that is, we can use the substitution $\{x \vdash f(y)\}$. A literal that has undergone a substitution from its original form is called an **instance** of that term.

5.2 Herbrand Sets

A **Herbrand Set** for a set of clauses C is an unsatisfiable set of ground clauses T such that for every clause $t \in T$ there is a clause in $c \in C$ such that t is an instance of c. A set of clauses C is unsatisfiable iff there is a Herbrand Set for C (**Herbrand's theorem**). This means that determining whether or not a Herbrand Set for C exists, consequently determines the satisfiability of C. Determining first-order satisfiability is now reduced to this *ground instantiation problem*, which lends itself much better to computation.

Our example is, in fact, satisfiable, meaning no Herbrand Set exists for its negation. Instead, we will use a constructed example. Given the set the clauses C:

$$\{\{Q(a)\}, \{P(f(x)), \neg Q(x)\}, \{\neg P(f(x))\}\}$$

A Herbrand Set for C is found after using the substitution $\{x \vdash a\}$:

$$\{\{Q(a)\}, \{P(f(a)), \neg Q(a)\}, \{\neg P(f(a))\}\}$$

We can see that this is unsatisfiable because the second clause can not be true unless either the first or third is false.

Given a set of clauses C, we can enumerate all ground clauses of C_g and check whether C_g is satisfiable. If C_g is unsatisfiable, then it is certain that C is unsatisfiable as well. While there are a countable infinite number of such clauses – eliminating the possibility of checking all of C_g at once, we can start with a set containing a small number of ground clauses C'_g. From there, we continually add one or more clause(s) to C'_g, check for satisfiability, and repeat these steps until C'_g is shown unsatisfiable. In fact, this method was outlined by Davis and Putnam [4].

To utilize this procedure for showing showing a first-order formula F is valid, F is negated ($\neg F$) and converted to clausal form. Then this set of clause – in addition to any other usable/relevant clauses C (axioms, relevant theorems, etc.) – is used as input to a proving procedure (such as the one described above). A Herbrand Set from the clausal form of $\neg F$ ($\wedge C$) exists (that is, $\neg F$ is unsatisfiable) iff $\neg F$ is not a satisfiable first-order formula. The unsatisfiability of $\neg F$ implies that F is valid.

It is worth noting that this method is only semi-decidable. If the set of clauses is satisfiable, the prover above will continue on forever, as it will never encounter an unsatisfiable set of ground clauses.$\neg F$ is used as the input because we (presumably) expect that F is valid.

5.3 DPLL

In the preceding section, we outlined a procedure for determining satisfiability from ground clauses. It became apparent, however that an efficient algorithm for determining satisfiability is essential if this method is ever to be practical. Each unique atom produced from a clause set's Herbrand Interpretation can be mapped to a unique propositional variable. Thus, our first-order validity problem is now further reduced to a classic CNF-SAT problem. Boolean satisfiability is a classic NP-Complete problem, so it follows that first-order satisfiability is also in the class NP.

While determining satisfiability is $O(2^n)$, these problems are often solvable within a reasonable amount of time in practice. The Davis-Putnam-Logemann-Loveland procedure (DPLL) [3] takes advantage of common heuristics to dynamically reduce the search space compared to a typical brute-force guess-and-check approach.

DPLL is backtracking search algorithm that implements unit propagation

and pure literal elimination. The algorithm is presented below [21], where Φ is a set of clauses:

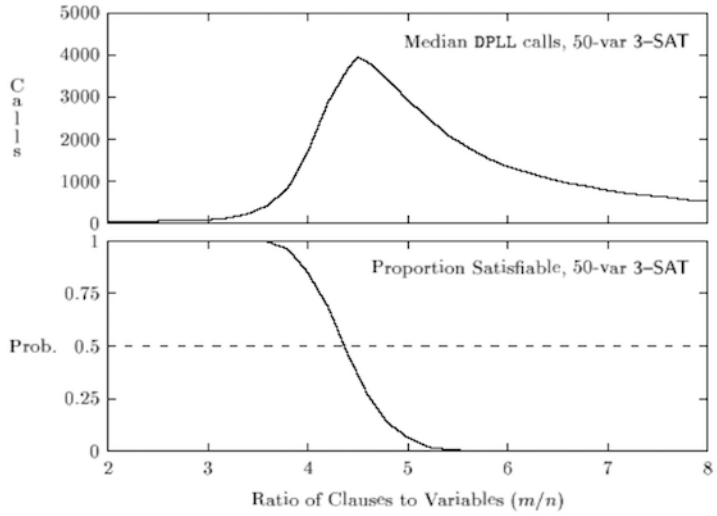
```
function DPLL(Φ)
    if Φ is a consistent set of literals
        then return true
    if Φ contains an empty clause
        then return false
    for every unit clause l in Φ
       Φ ← unit-propagate(l, Φ)
    for every literal l that occurs pure in Φ
       Φ ← pure-literal-assign(l, Φ)
    l ← choose-literal(Φ)
    return DPLL(Φ{l → ⊤}) or DPLL(Φ{l → ⊥})
```

A **unit clause** is a clause that contains only one literal l. In the case that there exists at least one unit clause, the `unit-propagate` procedure is called. This procedure removes all clauses containing l and removes all instances of $\neg l$ in the remaining clauses. The existence of a unit clause implies that the literal within that clause l must be *true*, meaning any other clause containing l is guaranteed to be true and any literals $\neg l$ are guaranteed to be false. Unit propagation improves the "goal-orientedness" of the algorithm by trimming the search space.

A **pure literal** is a literal whose complementary literal does not occur within any clauses in Φ. The `pure-literal-assign` procedure removes all clauses that contain a pure literal. It is worth noting that both `unit-propagate` and `pure-literal-assign` may spawn more unit clauses or pure literals, generally trimming the search space very quickly.

The `choose-literal` procedure is especially interesting. The literal chosen for the next DPLL essentially guides the search path. Assuming the input is satisfiable, a perfect selection of literals would theoretically reduce the runtime to linear. There has been extensive research in optimizing this step that will not be discussed here, so we'll pretend a literal is chosen non-deterministically.

The performance of DPLL is rarely close to exponential, but is instead very dependent on the number of clauses and variables. Evident below [2], DPLL finishes quickly when the ratio of clauses to variables is very low or very high, generally determining satisfiability and unsatisfiability respectively.



6 Growing the Clause Set

As reviewed earlier, ground clauses from the Herbrand Interpretation are incrementally added to some set C'_g until a Herbrand Set is found. Davis and Putnam suggested adding only one clause at a time before rechecking for satisfiability. Each single clause, however, is unlikely to change the outcome of such a check. To reduce the number of potentially expensive DPLL calls, multiple clauses are added between each call.

In practice, the ground clause set C'_g grows by including all clauses below an incrementing size-bound s. The size-bound then starts small, and grows incrementally until a Herbrand Set is (hopefully) found.

We will define the size s of terms t, literals L, clauses C, and clause sets as follows:

$$s(\neg L) = s(L)$$
$$s(P(t_1, ..., t_n)) = s(f(t_1, ..., t_n)) = 1 + s(t_1) + ... + s(t_n)$$
$$s(L_1 \vee ... \vee L_n) = s(L_1) + ... + s(L_n)$$
$$s(C_1 \wedge ... \wedge C_n) = s(C_1) + ... + s(C_n)$$
$$s(x) = s(c) = 1$$

From our prior example, we can calculate the size of each clause in C:

$$s(\{Q(a)\}) = 2$$
$$s(\{P(f(x)), \neg Q(x)\}) = 5$$
$$s(\{\neg P(f(x))\}) = 3$$

As is evident in the constructed domain of an example Herbrand Interpretation in section 5.1 ...

$$s(D) : \{s(a) = 1, s(f(a)) = 2, s(g(a)) = 2,$$
$$s(f(f(a))) = 3, s(g(f(a))) = 3, s(f(g(a))) = 3, ...\}$$

...a linear increase in the size-bound grows the ground clause set exponentially. Such explosion of clauses is useful in its early stages to quickly grow C'_g. Conversely, if the size-bound grows too large, the number of clauses below it may become too numerous, potentially filling all available memory and drastically slowing down the DPLL procedure.

6.1 An Observation

If we order the formation of terms from Herbrand Interpretation to be used in creating ground clauses in such a way...

$$H_0 : \{\text{all constants}\}$$
$H_1 : H_0 \cup \{\text{an application of each function to each permutation of terms in } H_0\}$
$H_n : H_{n-1} \cup \{\text{an application of each function to each permutation of terms in } \cup_{i=0}^{n-1} H_i\}$

...it becomes clear that terms are repeated many times as the size-bound grows. Using D as an example, we can see that every term in H_n is present multiple times in H_{n+1}.

$$H_0 : \{a\}$$
$$H_1 : \{a, f(a), g(a)\}$$
$$H_2 : \{a, f(a), g(a), f(f(a)), g(f(a)), f(g(a)), g(g(a))\}$$

Some terms above the size-bound may only be so large because of repeating sub-terms. As an example, $P(f(g(f(a))), f(g(f(a))))$ contains $f(g(f(a)))$ twice, "artificially" increasing the size of the entire term. Conventionally, if such a term was required in the proof, the proving procedure would not be able to terminate until the size-bound reached this term's size.

6.2 Redefining Size

We will redefine the size function s' to be exactly the same as stated previously, except repeated sub-terms are replaced with pointers to their first occurrence. Future occurrences of the sub-term have a size of 0. For demonstration, each sub-term will be assigned a number in order of appearance, and future occurrences will be replaced with that number. As an example:

$$C = \{P(f(a), g(f(a))), P(f(a), g(f(a)))\}$$

has a size $s(C) = 12$ using the formal definition. With our size reduction technique, we can rewrite C:

$$C = \{P(f(a), g(1)), 2\}$$

$s'(C) = 4$, much less than 12. In fact, for any clause L, $s'(L) \leq s(L)$. If this term was required for the proving procedure, it would become available once the size-bound reaches 4, instead of a drastically greater bound of 12.

Using s' grows the clause set much faster than the s, causing the same problems noted above to happen much faster. However, if clauses needed in the proof typically become available for use much faster using s', such a trade-off improve the overall efficiency of the proving procedure.

It should be noted that the pointers mentioned above could instead be assigned a size of 1. This change may slow the explosion of enumerated clauses under a given size-bound, as individual constants would never be reduced to a size of 0, regardless of repetitions.

7 Implementation

In order to more fully understand the impact this size reduction technique has on the enumeration of ground clauses, I have implemented the relevant structures – constants, variables, functions, predicates, and clauses – shown in Appendix A. With these, I have created a method for computing the size-reduced terms and an algorithm for enumerating clauses with a reduced size below a set size bound.

7.1 Notes about the Code

I chose to implement my project in Python as it allowed for fast prototyping and provided a simple work environment (with iPython Notebooks). Looking back, however, the class structure and strong typing of a language such as Java would have saved a lot of headaches in the long run.

Additionally, these structures were not built with efficiency in mind. To allow myself to work with higher levels of abstraction, namely with Python's sets, I altered all __hash__ methods to first serialize the object, then hash the resulting string. Identical objects are represented identically as strings, even if the objects are formed from different instances of objects. Two objects that should be the same hash to different values, but their string representations correctly hash to the same value. This workaround is especially useful given how often I use sets, but constant serialization has high overhead.

The structures I built have been connected with a Python implementation of the algorithms described in *Artifical Intellegence: A Modern Approach* [19], maintained on GitHub .

If shown useful, everything showcased here can be reimplemented much more efficiently within existing theorem proving infrastructure. The algorithm presented is well-suited for multithreading as each section of the search space can be explored separately.

7.2 Reduce Clause Size

The following code snippet takes a regular clause as input, and returns a new clause with repeated terms replaced with Repeat objects that have a size of 0. Calling $len()$ on the new clause returns the size of the clause based on the alternative s' definition.

```
def reduceSize(self):
    copy = deepcopy(self)
    subterms = set(copy.findSubterms())
    for i in range(len(copy.params)):
        subterm = copy.params[i]
        if subterm in subterms:
            subterms.remove(subterm)
            copy.params[i].reduceSize(subterms)
        else:
            copy.params[i] = Repeat(copy.params[i])
```

```
        return copy
```

This method, on its first pass, finds all sub-terms within the clause and stores them in the set `subterms`. On the second pass through the clause, whenever a sub-term is encountered, it is removed from `subterms`. If a sub-term is found that is not in `subterms`, it must be a repeat, and is replaced with an instance of `Repeat`, which has a size of 0.

As an example, we use the clause C from above to verify our findings. Instead of the numbering system described above, asterisks are applied around repeated terms for clarity.

```
print(clause, len(clause))
print(clause.reduceSize(), len(clause.reduceSize()))
```

```
{P(f(a),g(f(a))),P(f(a),g(f(a)))} 12
{P(f(a),g(*f(a)*)),*P(f(a),g(*f(a)*))*} 4
```

With this tool in the toolbox, I then developed an algorithm to enumerate all clauses with a reduced-size below a specified size-bound.

7.3 Enumerate Clauses

Constructing ground clauses below a specified size-bound is a relatively trivial procedure. With the altered definition of size, however, it becomes much more complex. There must be a way to "reach past" the given size-bound to explore larger terms that reduce down to be within the specified size. A naive approach would be to enumerate clauses well beyond the size-bound, check each of them, and add any that fit the criteria. This approach, however, requires checking a huge search space (shown by the exponential growth of ground clauses) of terms below a much larger (potentially infinite) size-bound – exactly the opposite of our goal.

Instead, the algorithm `enumerateClauses`, shown below (and further commented in Appendix B), enumerates all clauses recursively using the input clause – or clause set of the form described in Section 4 – as a starting point. Each variable is replaced with any available sub-term, forming a new clause as input to `enumerateClauses` again. Before creating this new clause, there is potential for a branch-and-bound check to trim the search space of incomplete clauses that are already above the size-bound, but it is dangerous as the replaced term may introduce other repeated subterms, thus reducing the overall size. This trimming is worth further investigation, but is not

implemented currently. If the clause is at the size-bound but not ground, each variable is replaced with already repeated terms (thus, not adding to the calculated reduced size). Ground clauses at or below the size-bound are added to the final set of clauses to be returned.

In order to preserve uniqueness of variable names, the `generalizeFunctions` takes functions as inputs and returns those functions with new (uniquely names) variables as parameters.

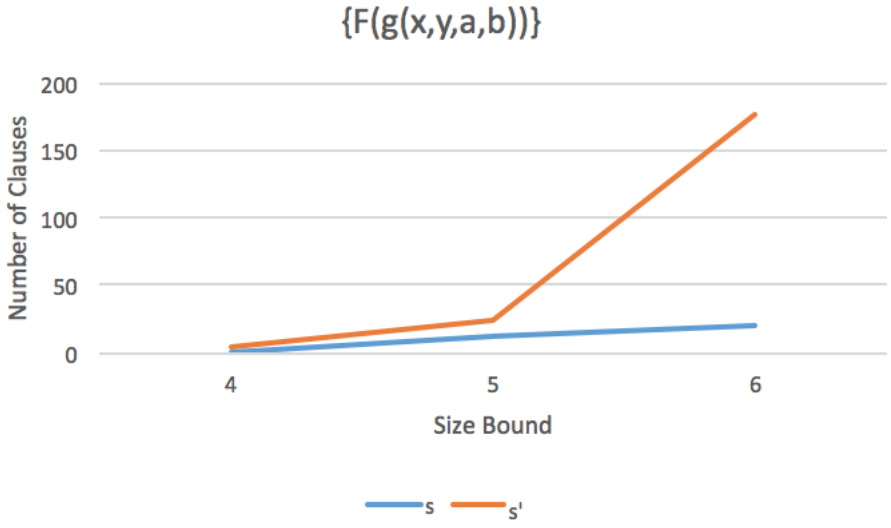
```
def enumerateClauses(clause, size, rpts = set(),
    ↪ clauseSet = set()):

    clauseReducedSize = len(clause.reduceSize())

    repeats = rpts.union(set(clause.findRepeats()))

    if clauseReducedSize>size:
        return clauseSet

    elif clause.isGround():
        clauseSet.add(clause)

    elif clauseReducedSize==size:
        for x in clause.findVariables():
            for i in repeats:
                enumerateClauses(clause.replace(x,i),
                    ↪ size, repeats, clauseSet)

    else:
        constants = set(clause.findConstants())
        variables = set(clause.findVariables())
        functions = generalizeFunctions(clause.
            ↪ findFunctions())
        for x in variables:
            for i in constants.union(repeats).union(
                ↪ functions):
                if (size-clauseReducedSize-len(i))>=0:
                    enumerateClauses(clause.replace(x,i
                        ↪ ), size, repeats, clauseSet)
```

```
return clauseSet
```

8 Example and Future Work

8.1 Example

The presented implementation relies too heavily on recursion to be written in a language with large stack frames such as Python. The original goal was to use example sentences that occur in the automatic proving testing problem set known as the Pelletier Problems [15]. Such examples are so large that the recursion depth limit is reached very quickly. Instead, below is a constructed example to showcase the difference in growth of clause sets between s and s'. Beyond a size-bound of 6, the recursion limit reached.



The largest clause C found by `enumerateClauses(clause, 6)` is:

$$\{F(g(g(g(b,a,a,b),g(b,a,a,b),a,b),g(g(b,a,a,b),g(b,a,a,b),a,b),a,b))\}$$

$s'(C) = 6$ while $s(C) = 30$. This means that we had to explore at least some part of the search space of size-bound 30. A size-bound beyond that quickly grows the search space to infeasibility.

8.2 Future Work

8.2.1 Problems with the Algorithm

Unfortunately, my implementation utilizes excessive recursion. While processing large (or very repetitive) clauses, the `enumerateClauses` procedure frequently hits the recursion depth limit, making this implementation infeasible in practice. To fix this problem, both `reduceSize` and `enumerateClauses` must be rewritten to better utilize iterative solutions. Given the formation of these structures, that would prove to be difficult. One potential improvement may be to port this algorithm to a functional programming language with proper tail call elimination. That being said, I have not been able to conceive of a way to rewrite the algorithm to be totally iterative. Obviously, the search space of all possible clauses is massive (growing much faster than that of the standard size definition). Even if this algorithm only needs to check some fraction of the whole space, it will still quickly surpass a size that can reasonably be searched with current computers. Because of this, I am not convinced that exploiting the size-bound is a feasible method for improving the efficiency of first-order provers – but I am excited to be proven wrong.

8.2.2 Assuming the Problems are Fixed

If, at some point, a method for this alternative enumeration of clauses is found, more experiments would be required still to determine if such a method is worth the additional cost. More testing would be required to decide if the additional processing time to find these clauses is less than the time saved from including them earlier.

9 Conclusion

Redefining the definition of size has shown to exponentially increase the number of clauses under a specified size-bound. Unfortunately, the enumeration of such clauses requires reaching far into the search space of a large standardly-define size-bound. By recursively building clauses with `enumerateClauses`, much of that extended search space is not relevant and, thus, not explored. Regardless, the required computing power of this procedure very quickly surpasses that of a typical commercial computer. It is

possible such a method could run properly on a more powerful machine. Even then, however, more experimentation would be necessary to evaluate the efficacy of this method compared to the current standard. The presented code has, at minimum, presented a proof of concept for future study.

References

[1] E. M. Clarke, M. Khaira, and X. Zhao. Word level model checking - avoiding the pentium fdiv error. In *Proceedings of the 33rd Annual Design Automation Conference*, DAC '96, pages 645–648, New York, NY, USA, 1996. ACM.

[2] S. A. Cook and D. G. Mitchell. Finding hard instances of the satisfiability problem. In *Satisfiability Problem: Theory and Applications: DIMACS Workshop, March 11-13, 1996*, volume 35, page 1. American Mathematical Soc., 1997.

[3] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[4] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[5] G. Frege. From frege to gödel: a source book in mathematical logic, 1879-1931. In J. Van Heijenoort, editor, *Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought*, volume 9. Harvard University Press, 1967.

[6] G. Gentzen. Untersuchungen über das logische schließen. i. *Mathematische zeitschrift*, 39(1):176–210, 1935.

[7] P. C. Gilmore. A proof method for quantification theory: Its justification and realization. *IBM Journal of Research and Development*, 4(1):28–35, Jan 1960.

[8] L. Kovács and A. Voronkov. First-order theorem proving and vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.

[9] J. McCarthy. *Programs with common sense*. RLE and MIT Computation Center, 1960.

[10] W. McCune. *Otter 3.0 reference manual and guide*, volume 9700. Argonne National Laboratory Argonne, IL, 1994.

[11] W. McCune. Solution of the robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[12] W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[13] O. Owen. *The Organon: Or Logical Treatises of Aristotle : with the Introduction of Porphyry.* Number v. 1 in Bohn's classical library. G. Bell, 1901.

[14] G. Peano. From frege to gödel: a source book in mathematical logic, 1879-1931. In J. Van Heijenoort, editor, *The principles of arithmetic, presented by a new method*, volume 9. Harvard University Press, 1967.

[15] F. J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of automated reasoning*, 2(2):191–216, 1986.

[16] E. Post. From frege to gödel: a source book in mathematical logic, 1879-1931. In J. Van Heijenoort, editor, *Introduction to a general theory of elementary propositions*, volume 9. Harvard University Press, 1967.

[17] D. Prawitz. An improved proof procedure. *Theoria*, 26(2):102–139, 1960.

[18] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, Jan. 1965.

[19] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Higher Ed, 2016.

[20] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In *International Conference on Automated Deduction*, pages 341–355. Springer, 1994.

[21] Wikipedia. Dpll algorithm — wikipedia, the free encyclopedia, 2017. [Online; accessed 30-March-2017].

A Structure Implementations

```
#deepcopy used to create repeated terms while keeping
   ↪ the original (that is possibly used elsewhere)
   ↪ intact.
from copy import deepcopy

#The structures are connected to a aima logic library
from utils import *
from logic import *

#Implementation of constant.
#Overrides str, len, eq, and hash.
class Constant:
    registry = set()

    def __init__(self, s):
        self.sym = s
        self.arity = 0
        self.length = 1
        Constant.registry.add(self)

    def __str__(self):
        return self.sym

    def __len__(self):
        return self.length

    def __eq__(self, other):
        return str(self) == str(other)

    def __hash__(self):
        return hash(str(self))

    #Any repeats are replaced by Repeats.
    def reduceSize(self, subterms):
        subterm = self
```

```
            if subterm in subterms:
                subterms.remove(subterm)
            else:
                self = Repeat(self)

    def reducedSize(self):
        return self.length

        #Will always be a ground term.
        #Base of higher level isGround recursion.
        def isGround(self):
            return True

        #No variables.
        #Returns empty list to conform with other
           ↪ structures use of list concacts.
        def findVariables(self):
            return []

        #Is a constants.
        #Returns object wrapped in list to conform with
           ↪ other structures use of list concacts.
        #Base of higher level findConstants recursion.
        def findConstants(self):
            return [self]

        #Is inherently a repeated term once found.
        #Base of higher level findRepeats recursion.
        def findRepeats(self):
            return [self]

        #No functions.
        #Returns empty list to conform with other
           ↪ structures use of list concacts.
        def findFunctions(self):
            return []

        #Is a subterm.
```

```
    #Base of higher level findSubterms recursion.
    def findSubterms(self):
        return {self}

    #Nothing to replace.
    #Base of higher level replace recursion.
    def replace(self, var, rpt):
        return self

    #Once we enumerate clauses, we must prepare them
    #to be in the correct form for the imported DPLL
       ↪ implementation.
    def prepare(self):
        return Symbol(str(self))

#Implementation of repeat.
#Encapsulates replaced object in case the information
   ↪ is needed.
#Length is always 0.
#Overrides str, len, eq, and hash
class Repeat:

    def __init__(self, original):
        self.sym = '*'
        self.origin = original
        self.length = 0

    #Currently shows the repeated term surrounded by
       ↪ stars.
    #Switch commented line for more informative printed
       ↪  information.
    def __str__(self):
        return self.sym + str(self.origin) + self.sym
    #return self.origin.sym

    def __len__(self):
        return self.length
```

```
def __eq__(self, other):
    return (str(self) == str(other)) or (str(self.
        ↪ origin) == str(other))

def __hash__(self):
    return hash(str(self.origin))

#Size already reduced.
#Implemented so that reduceSize still works if
    ↪ accidentally called multiple times.
def reduceSize(self):
    pass

def reducedSize(self):
    return self.length

#Ground determined by encapsulated object.
#Passed on to that function.
def isGround(self):
    return self.origin.isGround()

#Variables determined by encapsulated object.
#Passed on to that function.
def findVariables(self):
    return self.origin.findVariables()

#Constants determined by encapsulated object.
#Passed on to that function.
def findConstants(self):
    return self.origin.findConstants()

#First instance of this repeat already included in
    ↪ search.
#Empty list used to conform with higher level
    ↪ findRepeat.
def findRepeats(self):
    return []
```

```
    #Functions determined by encapsulated object.
    #Passed on to that function.
    def findFunctions(self):
        return self.origin.findFunctions()

    #Is a not a subterm. Original will be kept, not
       ↪ repeats..
    #Base of higher level findSubterms recursion.
    def findSubterms(self):
        return {}

    #No variable to replace.
    #Should not replace underlying object (if possible)
    #because there is some clause that does not have
       ↪ this as a repeat.
    def replace(self, var, rpt):
        return self

#Repeats will never be in any of the final clauses,
#so there is no need for prepare().

#Implementaion of variable.
#Frequently replaced with other objects in enumerate
   ↪ algorithm.
#Overrides str, len, eq, and hash.
class Variable:

    tally = 0

    def __init__(self, s=0):
        if s==0:
            self.sym = self.name()
        else:
            self.sym = s
        #Not self.length = 1 Use minimal possible size.
        self.length = 0

    def __str__(self):
```

```
        return self.sym

    def __len__(self):
        return self.length

    def __eq__(self, other):
        return str(self) == str(other)

    def __hash__(self):
        return id(str(self))

    #Special naming used for generalizeFunctions() to
       ↪ ensure unique names.
    def name(self):
        Variable.tally+=1
        return str(Variable.tally)

    #Any repeats are replaced by Repeats.
    def reduceSize(self, subterms):
        subterm = self
        if subterm in subterms:
            subterms.remove(subterm)
        else:
            self = Repeat(self)

    def reducedSize(self):
        return self.length

    #Will never be a ground term.
    #Base of higher level isGround recursion.
    def isGround(self):
        return False

    #Is a variable.
    #Returns object wrapped in list to conform with
       ↪ other structures use of list concacts.
    #Base of higher level findVariables recursion.
    def findVariables(self):
```

```
        return [self]

    #No variables.
    #Returns empty list to conform with other
       ↪ structures use of list concacts.
    def findConstants(self):
        return []

    #Is Inherently a repeated term once found.
    #Base of higher level findRepeats recursion.
    def findRepeats(self):
        return []

    #No functions.
    #Returns empty list to conform with other
       ↪ structures use of list concacts.
    def findFunctions(self):
        return []

    #Is a subterm.
    #Base of higher level findSubterms recursion.
    def findSubterms(self):
        return {self}

    #Returns a copy of the predicate with 'var'
       ↪ replaced with 'rpt'.
    #Should not need to copy, because it was done by
       ↪ the clause.
    def replace(self, var, rpt):
        #variable = deepcopy(self)
        variable = self
        if self==var:
            return rpt
        else:
            return variable

#Variables will never be in any of the final clauses,
#so there is no need for prepare().
```

```
#Implementation of function.
#Includes symbol (must be unique) and list of
   ↪ paramaters.
#Registry containts copies of all functions that may be
   ↪  altered in enumerate algorithm.
#Consider including arity parameter.
#Overrides str, len, eq, and hash.
class Function:

    #Length calculated by definition: 1 + length of
       ↪ each parameter.
    def __init__(self, s, *args):
        self.sym = s
        self.params = list(args)
        self.arity = len(args)
        self.length = 1+sum([len(i) for i in self.
           ↪ params])

    def __str__(self):
        toReturn = self.sym + "("
        for i in self.params:
            toReturn += str(i) + ","
        toReturn = toReturn[:-1] + ")"
        return toReturn

    def __len__(self):
        return 1+sum([len(i) for i in self.params])

    def __eq__(self, other):
        return str(self) == str(other)

    def __hash__(self):
        return hash(str(self))

    #Any repeats are replaced by Repeats.
    #Then reduces size of each parameter.
    def reduceSize(self, subterms):
```

```
        for i in range(len(self.params)):
            subterm = self.params[i]
            if subterm in subterms:
                subterms.remove(subterm)
                self.params[i].reduceSize(subterms)
            else:
                self.params[i] = Repeat(self.params[i])

    #Scans each parameter.
    #If any are not ground, the entire instance of the
       ↪ function is not ground.
    def isGround(self):
        for i in self.params:
            if not i.isGround():
                return False
        return True

    #Scans each parameter.
    #Finds variables in each parameter and adds all of
       ↪ them to the list.
    def findVariables(self):
        variables = []
        for i in self.params:
            variables = variables + i.findVariables()
        return variables

    #Scans each parameter.
    #Finds constants in each parameter and adds all of
       ↪ them to the list.
    def findConstants(self):
        constants = []
        for i in self.params:
            constants = constants + i.findConstants()
        return constants

    #Scans each parameter.
    #Finds repeats in each parameter and adds all of
       ↪ them to the list.
```

```
def findRepeats(self):
    repeats = []
    for i in self.params:
        repeats = repeats + i.findRepeats()
    if self.isGround():
        repeats.append(self)
    return repeats

#Is inherently a function.
#Finds functions in each parameter and adds all of
   ↪ them to the list.
def findFunctions(self):
    functions = [self]
    for i in self.params:
        functions = functions + i.findFunctions()
    return functions

#Is a subterm.
#Finds all subterms in each parameter and adds them
   ↪  to the list.
def findSubterms(self):
    subterms = {self}
    for i in self.params:
        subterms = subterms.union(i.findSubterms())
    return set(subterms)

#Returns a copy of the predicate with 'var'
   ↪ replaced with 'rpt'.
#Should not need to copy, because it was done by
   ↪ the clause.
def replace(self, var, rpt):
    #function = deepcopy(self)
    function = self
    for i in range(len(function.params)):
        term = function.params[i]
        if term==var:
            function.params[i] = rpt
        else:
```

```
                function.params[i] = term.replace(var,
                    ↪ rpt)
        return function

    #Once we enumerate clauses, we must prepare them to
        ↪ be in the correct form for the imported DPLL
        ↪ implementation.
    def prepare(self):
        return Expr(self.sym, *[i.prepare() for i in
            ↪ self.params])

#Implementation of predicate (very similar
    ↪ implementation to function).
#Includes symbol (must be unique) and list of
    ↪ paramaters.
#Overrides str, len, eq, and hash
class Predicate:

    #Length calculated by definition: 1 + length of
        ↪ each parameter.
    def __init__(self, s, t=True, *args):
        self.sym = s
        self.params = list(args)
        self.length = 1+sum([len(i) for i in self.
            ↪ params])
        self.true = t

    def __str__(self):
        toReturn = ""
        if not self.true:
            toReturn += '~'
        toReturn += self.sym + "("
        for i in self.params:
            toReturn += str(i) + ","
        toReturn = toReturn[:-1] + ")"
        return toReturn

    def __len__(self):
```

```
        return 1+sum([len(i) for i in self.params])

    def __eq__(self, other):
        return str(self) == str(other)

    def __hash__(self):
        return hash(str(self))

    #Any repeats are replaced by Repeats.
    #Then reduces size of each parameter.
    def reduceSize(self, subterms):
        for i in range(len(self.params)):
            subterm = self.params[i]
            if subterm in subterms:
                subterms.remove(subterm)
                self.params[i].reduceSize(subterms)
            else:
                self.params[i] = Repeat(self.params[i])

#Scans each parameter.
#If any are not ground, the entire instance of the
   ↪ function is not ground.
def isGround(self):
    for i in self.params:
        if not i.isGround():
            return False
        return True

    #Scans each parameter.
    #Finds variables in each parameter and adds all of
       ↪ them to the list.
    def findVariables(self):
        variables = []
        for i in self.params:
            variables = variables + i.findVariables()
        return variables

    #Scans each parameter.
```

```
#Finds constants in each parameter and adds all of
   ↪ them to the list.
def findConstants(self):
    constants = []
    for i in self.params:
        constants = constants + i.findConstants()
    return constants

#Scans each parameter.
#Finds repeats in each parameter and adds all of
   ↪ them to the list.
def findRepeats(self):
    repeats = []
    for i in self.params:
        repeats = repeats + i.findRepeats()
    if self.isGround():
        repeats.append(self)
    return repeats

#Scans each parameter.
#Finds functions in each parameter and adds all of
   ↪ them to the list.
def findFunctions(self):
    functions = []
    for i in self.params:
        functions = functions + i.findFunctions()
    return functions

#Is a subterm.
#Finds all subterms in each parameter and adds them
   ↪  to the list.
def findSubterms(self):
    subterms = {self}
    for i in self.params:
        subterms = subterms.union(i.findSubterms())
    return set(subterms)

#Returns a copy of the predicate with 'var'
```

```
        ↪ replaced with 'rpt'.
    #Should not need to copy, because it was done by
        ↪ the clause.
    def replace(self, var, rpt):
        predicate = self
        for i in range(len(predicate.params)):
            term = predicate.params[i]
            if term==var:
                predicate.params[i] = rpt
            else:
                predicate.params[i] = term.replace(var,
                    ↪ rpt)
        return predicate

    #Once we enumerate clauses, we must prepare them to
        ↪ be in the correct form for the imported DPLL
        ↪ implementation.
    def prepare(self):
        if self.true:
            return Expr(self.sym, *[i.prepare() for i
                ↪ in self.params])
        else:
            return Expr('~', Expr(self.sym, *[i.prepare
                ↪ () for i in self.params]))

#Implementation of clause (very similar implementation
    ↪ to function and predicate).
#Includes list of paramaters.
#Overrides str, len, eq, and hash
class Clause:

    #Length calculated by definition: sum of length of
        ↪ each parameter.
    def __init__(self, *args):
        self.params = list(args)
        self.length = sum([len(i) for i in self.params
            ↪ ])
```

```
def __str__(self):
    toReturn = "{"
    for i in self.params:
        toReturn += str(i) + ","
    toReturn = toReturn[:-1] + "}"
    return toReturn

def __len__(self):
    return sum([len(i) for i in self.params])

def __eq__(self, other):
    return str(self) == str(other)

def __hash__(self):
    return hash(str(self))

#First pass, finds all subterms within the clause.
#Second pass, removes subterms when encountered.
#Any subterm encountered that is not in the set is
   ↪ a repeat.
#Repeats are replaced by Repeats.
def reduceSize(self):
    copy = deepcopy(self)
    subterms = set(copy.findSubterms())
    for i in range(len(copy.params)):
        subterm = copy.params[i]
        if subterm in subterms:
            subterms.remove(subterm)
            copy.params[i].reduceSize(subterms)
        else:
            copy.params[i] = Repeat(copy.params[i])
    return copy

#Scans each parameter.
#If any are not ground, the entire instance of the
   ↪ function is not ground.
def isGround(self):
    for i in self.params:
```

```
            if not i.isGround():
                return False
        return True

    #Scans each parameter.
    #Finds variables in each parameter and adds all of
       ↪ them to the list.
    def findVariables(self):
        variables = []
        for i in self.params:
            variables = variables + i.findVariables()
        return variables

    #Scans each parameter. Finds constants in each
       ↪ parameter and adds all of them to the list.
    #Herbrand structures require at least one constant.
    #If there are no constants, one arbitrary constant
       ↪ 'C' is added.
    def findConstants(self):
        constants = []
        for i in self.params:
            constants = constants + i.findConstants()
        if not constants:
            constants.append(Constant('C'))
        return constants

    #Scans each parameter.
    #Finds repeats in each parameter and adds all of
       ↪ them to the list.
    #Unlike other structures, does not include itself.
    def findRepeats(self):
        repeats = []
        for i in self.params:
            repeats = repeats + i.findRepeats()
        return repeats

    #Scans each parameter.
    #Finds functions in each parameter and adds all of
```

```
    ↪ them to the list.
def findFunctions(self):
    functions = []
    for i in self.params:
        functions = functions + i.findFunctions()
    return functions

#Is kind of a subterm - no harm in including it.
#Finds all subterms in each parameter and adds them
    ↪  to the list.
def findSubterms(self):
    subterms = {self}
    for i in self.params:
        subterms = subterms.union(i.findSubterms())
    return set(subterms)

#Returns a copy of the clause with 'var' replaced
#with 'rpt' to be used in deeper levels of
    ↪ recursion.
def replace(self, var, rpt):
    clause = deepcopy(self)
    for i in range(len(clause.params)):
        term = clause.params[i]
        if term==var:
            clause.params[i] = rpt
        else:
            clause.params[i] = term.replace(var,
                ↪ rpt)
    return clause

#Once we enumerate clauses, we must prepare them
#to be in the correct form for the imported DPLL
    ↪ implementation.
def prepare(self):
    if len(self.params)>1:
        return Expr('|', *[i.prepare() for i in
            ↪ self.params])
    else:
```

```
            return self.params[0].prepare()

def prepareDPLL(s):
    if len(s)>1:
        return Expr('&', *[i.prepare() for i in s])
    else:
        return s.pop().prepare()
```

B Enumeration Algorithm

```
"""
    Function:
    enumerateClauses

    Use:
    Given a clause and sizebound, enumerates all ground
       ↪  clauses whose s_dag - calculated by len(
       ↪ reduceSize()) -
    is less than the desired sizebound These results
       ↪ can then be fed into a DPLL implementation to
       ↪  determine
    satisfiability.

    Inputs:
    clause (Clause): The desired clause for ground
       ↪ instances under a sizebound to be enumerated
       ↪ from.
    size (int): The sizebound. Conventionally compared
       ↪ to s_lin. Compared to s_dag here.
    rpts (set): The set of repeats. Is empty to start,
       ↪ used within algorithm. Do not pass anything
       ↪ here.
    clauseSet (set): The set of clauses that are both
       ↪ ground and under the sizebound.
    Is empty to start, used within algorithm. Do not
       ↪ pass anything here.

    Output:
    clauseSet (set): The set of clauses that are both
       ↪ ground and under the sizebound after the
       ↪ terminating condition.

    Helper Functions:
    generalizeFunctions:
    Returns a copy of a function with all terms
```

```
        ↪ replaced by new variables.
    Inputs:
    functions (set): a set of functions.
    Outputs:
    genfunctions (set): the same set of functions with
        ↪ all terms replaced with new variables.

    """

def enumerateClauses(clause, size, rpts = set(),
    ↪ clauseSet = set()):

    #Reduced clause length. Instead of recalculating
        ↪ often, we'll calculate it once and save it.
    clauseReducedSize = len(clause.reduceSize())

    #Create an effective closure around repeats, (
        ↪ potentially) growing and shrinking with
        ↪ recursion depth.
    repeats = rpts.union(set(clause.findRepeats()))

    #Base Case: Clause size is larger than size bound.
        ↪ Does nothing to clauseSet and returns.
    if clauseReducedSize>size:
        return clauseSet

    #Base Case: The clause is ground. Add it to
        ↪ clauseSet and return.
    elif clause.isGround():
        clauseSet.add(clause)
        return clauseSet

    #Case 1: The clause has reached the sizebound, but
        ↪ is not ground. Recursively call
        ↪ enumerateClauses with only
    #terms of size 0 - repeats.
    elif clauseReducedSize==size:
        for x in clause.findVariables():
```

```
            for i in repeats:
                enumerateClauses(clause.replace(x,i),
                    ↪ size, repeats, clauseSet)
        return clauseSet

    #Case 2: The clause has not reached the sizebound
        ↪ and is not ground. Recursively call
        ↪ enumerateClauses with
    #any possible goal-oriented term - constants,
        ↪ functions, or repeats.
    else:
        constants = set(clause.findConstants())
        variables = set(clause.findVariables())
        functions = generalizeFunctions(clause.
            ↪ findFunctions())
        for x in variables:
            for i in constants.union(repeats).union(
                ↪ functions):
                #Some sort of branch-and-bound approach
                    ↪  to cut unnecessary
                    ↪ enumerateClauses calls may be
                    ↪ possible here.
                enumerateClauses(clause.replace(x,i),
                    ↪ size, repeats, clauseSet)
        return clauseSet

    return clauseSet

def generalizeFunctions(functions):
    genfunctions = set()
    for fn in functions:
        genfn = deepcopy(fn)
        for i in range(len(genfn.params)):
            if isinstance(genfn.params[i], Variable):
                genfn.params[i] = Variable()
        genfunctions.add(genfn)
    return genfunctions
```